Computergestütztes wissenschaftliches Rechnen
SoSe 2004

Alexander K. Hartmann, Universität Göttingen

18. Juni 2004

7.5 Heaps

Laufzeit des Programms:

Anzal der Stöße pro Zeiteinheit: $O(N)$
Suche des nächsten Ereignisses: $O(N)$
$\Rightarrow O(N^2) =$ "langsam".

Verbesserung: $O(N \log N)$, wenn man einen Heap verwendet.

Vorschau:
Lauzeitbeispiel: $N = 500, t_{\text{end}} = 10000$.

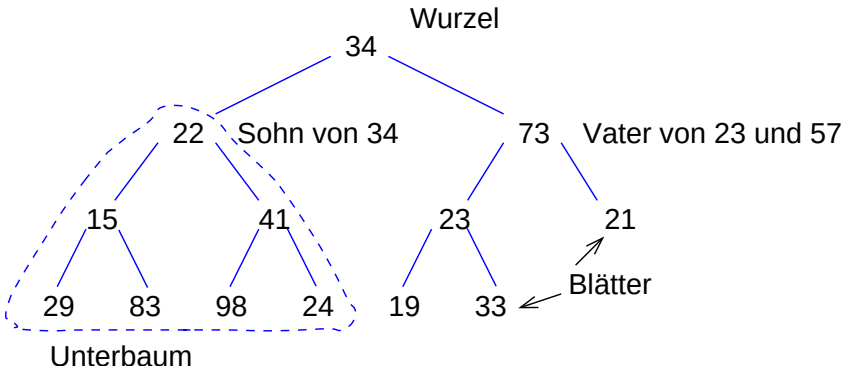
```
time chain 500 10000
21.36user 0.07system 0:21.70elapsed 98%CPU (0avgtext+0avgdata 0maxresident)k
0inputs+0outputs (133major+20minor)pagefaults 0swaps
```

```
time chain_heap 500 10000
7.92user 0.01system 0:08.08elapsed 98%CPU (0avgtext+0avgdata 0maxresident)k
0inputs+0outputs (133major+23minor)pagefaults 0swaps
```

mit Heap → schnellere Programm → größere Systeme ($N = 16383$ zu $N = 1281$)
→ verlässlichere, ANDERE Ergebnisse [1] (Crossover).

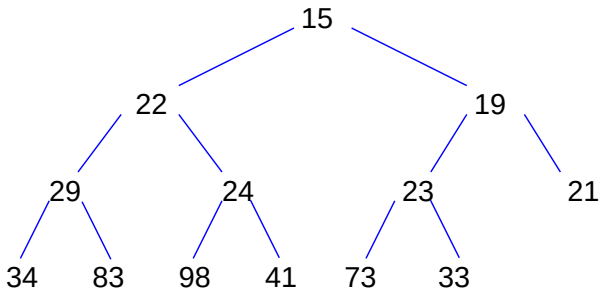
Heaps = Spezialfall von Bäumen.

Bäume = hierarchisch angeordnete Elemente (*Knoten*), z.B. Stammbäume. Hier Elemente=Zahlen.



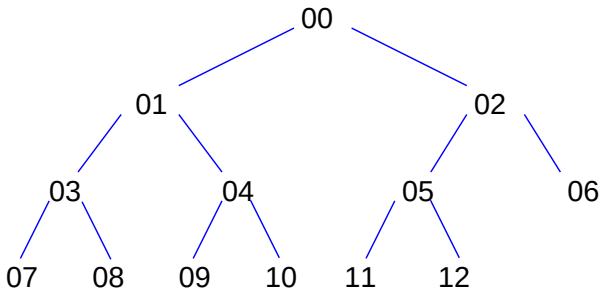
Binäre Bäume: jeder Knoten hat maximal zwei direkte Nachfolger (=Söhne)
Vorgänger der Knoten = Väter
Würzel = Knoten auf höchster Ebene
Blätter = Knoten ohne Nachfolger
Teilbaum = alle Nachfolger eines Knotens (=Wurzel des Teilbaums)

Heap = teilgeordneter Baum, der für jeden Unterbaum das (hier) kleinste Element an der Wurzel stehen hat
→ jedes Element ist kleiner als seine Söhne
Bsp:



Damit: das erste Element ist IMMER das kleinste, also z.B. das nächste Ereignis
→ schneller Zugriff ($O(1)$).

Realiserung von Bäumen: z.B. Elemente, die durch Zeiger verknüpft sind.
Für Heaps: effiziente Realisierung als Array:



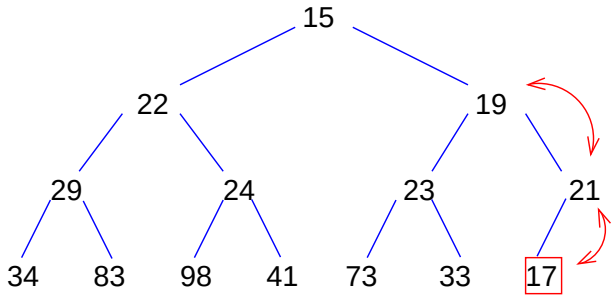
Knoten i:
Vater: $(i-1)/2$ (`int` Operation)
linker Sohn: $2i+1$
rechter Sohn: $2i+2$

Grundlegende Heap Operationen:
Einfügen:

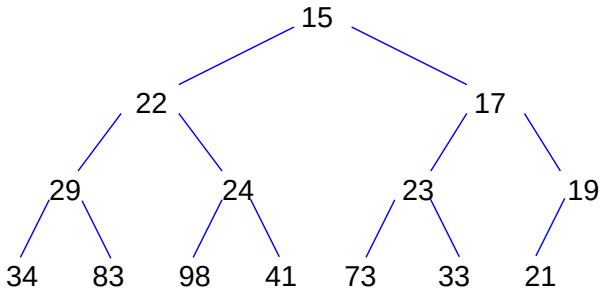
algorithm heap_insert()
begin
    füge Element am Ende hinzu;
    while (Element kleiner als Vater)
        vertausche mit Vater;
end

(siehe `heap_insert()` in `chain_cheap.c`)

Bsp: Einfügen von "17"



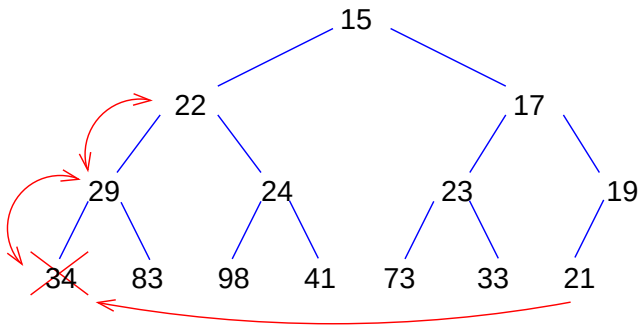
ergibt



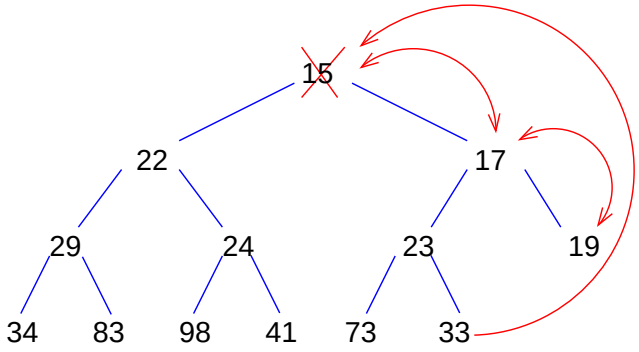
maximal ein Durchlauf von einem Blatt zur Wurzel
$\rightarrow$ Zeit $O(\log N)$
Entfernen:

algorithm heap_remove()
begin
ersetze Element durch letztes Element;
if (Element kleiner als Vater) **then**
while (Element kleiner als Vater)
vertausche mit Vater;
else
while (Element größer als ein Sohn)
vertausche mit kleinerem Sohn;
end

Fall A:



Fall B:



→ Zeit $O(\log N)$
Implementierungshinweis: Es werden auch Ereignisse mitten aus dem Heap entfernt (wenn sich die Zeiten benachbarter Ereignisse ändern).
→ damit das schnell geht, wird für jedes Ereignis in dem nach Ort geordnetem Ereignis Array auch seine Position im Heap gespeichert. (siehe Typ `heap_elem_t` in `chain_heap.c`). Diese Position muß bei jeder Verschiebung im Heap mit aktualisiert werden. (Ohne diese Abspeicherung müsste wieder der ganze Heap durchsucht werden, wenn ein Ereignis mitten aus dem Heap entfernt wird → wieder $O(N)$). Solche "Doppelverweise" (hier Heap → Array, Array → Heap) sind oft nötig, wenn man effiziente Programme schreiben will.
(siehe `heap_remove()` in `chain_cheap.c`)

Zugriff auf das erste Element im Heap: $O(1)$ (im Vergleich zu $O(N)$ bei der einfachen Implementierung).
→ Gesamtlaufzeit $O(N \log N)$.

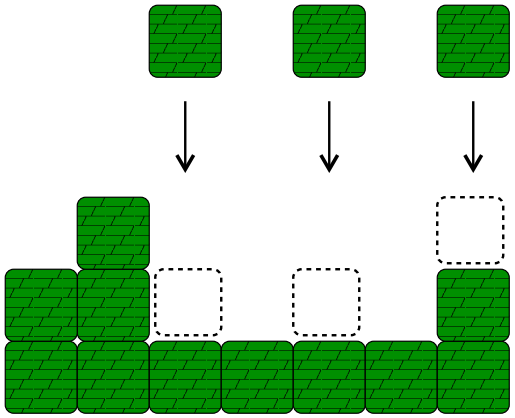
8 Monte Carlo Simulationen

heißt: Simulation mit (Pseudo-) Zufallszahlen. Dient zur Modellierung statistischer Prozesse. Hier als Beispiele: Wachsen von Oberflächen durch Molekularstrahlepitaxie (MBE) und Verhalten von Ising Ferromagneten.

8.1 Molekularstrahlepitaxie

Wachsen von molekularen Schichtstrukturen, z.B. für Halbleiterstrukturen. Atome werde thermisch "verdampft" und fallen langsam auf Substrat. Durch Masken + Wechsel der Atomsorten → Strukturierung [2].

Modell I:
Zufallsdeposition. Atome = Kästchen. Fallen senkrecht an zufällig ausgewählten Orten runter und bleiben beim Auftreffen auf Oberfläche liegen:



Beschreibung durch diskretes Höhenfeld $h(\underline{x})$: Bei Anlagerung: $h(\underline{x}) = h(\underline{x})+1 \rightarrow$ SOS Modell (Solid-on-Solid) $\rightarrow$ keine Überhänge oder Gitterfehler. Hier: 1-dim $\rightarrow$ Feld $h[0] \ldots h[L-1]$ (L =Systemgröße)

Unterprogramm für Zufallsdeposition:

```
/****************** random_deposition() **************/
/** Deposes particles randomly on the 1d surface    **/
/** given by 'h'. One sweep is performed            **/
/** PARAMETERS: (*)= return-parameter               **/
/**       glob: global data                         **/
/**     (*)  h: height field                        **/
/** RETURNS:                                        **/
/**    nothing                                      **/
/*****************************************************/
void random_deposition(system_t *glob, int *h)
{
  int pos;          /* position on x-axis of surface */
  int step;

  for(step=0; step<glob->L; step++)
  {
    pos = (int) floor(glob->L*drand48());
    h[pos]++;
  }
}
```

Messung der Monte-Carlo Zeit in *Sweeps* t = Anzahl der zufälligen Atome geteilt durch Fläche des Substrates.
Ergebnis:

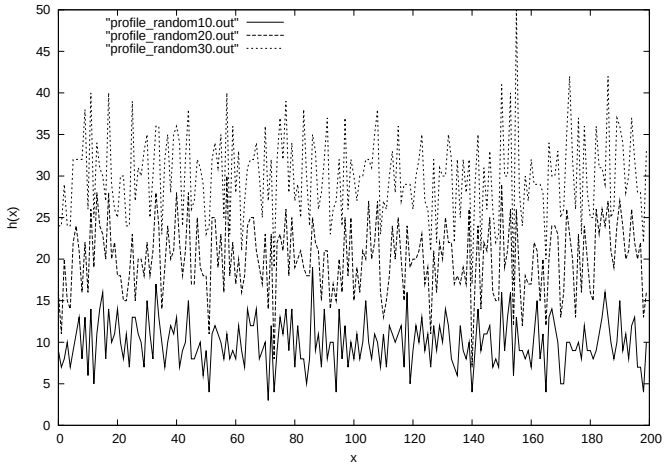


Abbildung 1: Höhenprofile für Zufallsdeposition für nach drei verschiedenen Zeiten $t = 10, 20, 30$, Systemgröße $L = 200$.

Literatur

[1] P. Grassberger, W. Nadler, and Lei Yang, Phys. Rev. Lett. **89**, 180601 (2002).

[2] A.-L. Barabasi and H.E. Stanley, Fractal Concepts in Surface Growth, (Cambridge University Press, 1995).